UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/781,867	02/20/2004	Edward Colles Nevill	550-513	5181

23117 7590 08/09/2007
NIXON & VANDERHYE, PC
901 NORTH GLEBE ROAD, 11TH FLOOR
ARLINGTON, VA 22203

EXAMINER	
MCFADDEN, MICHAEL B	
ART UNIT	**PAPER NUMBER**
2188	

MAIL DATE	DELIVERY MODE
08/09/2007	PAPER

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

UNITED STATES PATENT AND TRADEMARK OFFICE

Commissioner for Patents
United States Patent and Trademark Office
P.O. Box 1450
Alexandria, VA 22313-1450
www.uspto.gov

BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

Application Number: 10/781,867
Filing Date: February 20, 2004
Appellant(s): NEVILL, EDWARD COLLES

Stanley C. Spooner
For Appellant

EXAMINER'S ANSWER

This is in response to the appeal brief filed 12 April 2007 appealing from the Office action mailed 20 October 2006.

(1) Real Party in Interest

A statement identifying by name the real party in interest is contained in the brief.

(2) Related Appeals and Interferences

The examiner is not aware of any related appeals, interferences, or judicial proceedings which will directly affect or be directly affected by or have a bearing on the Board's decision in the pending appeal.

(3) Status of Claims

The statement of the status of claims contained in the brief is correct.

(4) Status of Amendments After Final

The appellant's statement of the status of amendments after final rejection contained in the brief is correct.

(5) Summary of Claimed Subject Matter

The summary of claimed subject matter contained in the brief is correct.

(6) Grounds of Rejection to be Reviewed on Appeal

The appellant's statement of the grounds of rejection to be reviewed on appeal is correct.

(7) Claims Appendix

The copy of the appealed claims contained in the Appendix to the brief is correct.

(8) Evidence Relied Upon

Gupta et al., "Turbo-charging Java HotSpot Virtual Machine, v1.4.x to Improve the Performance and Scalability of Application Servers",

http://java.sun.com/developer/technicalArticles/Programming/turbo/, Sun Microsystems, Inc., (archive.org date of Dec 08, 2003), pp.1-17.

(9) Grounds of Rejection

The following ground(s) of rejection are applicable to the appealed claims:

Claim Rejections - 35 USC § 102

1. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless –
>
> (b) the invention was patented or described in a printed publication in this or a foreign country or in public use or on sale in this country, more than one year prior to the date of application for patent in the United States.

2. Claims 1-6, 9, 11-16, 19, 21-26, and 29 are rejected under 35 U.S.C. 102(b) as being anticipated by Wilson ("Uniprocessor Garbage Collection Techniques").

3. **Regarding Claim 1, 11, and 21,** Wilson discloses a method of controlling execution of a processing task within a processing system, said method comprising the steps of: executing said processing task including allocating memory areas for data storage, and suspending an actual execution path of said processing task at an execution point to perform memory management said memory management comprising the steps of: identifying at least one data item roots occurring in the course of execution and accessible to said processing task at said execution point which specify reference values pointing to respective ones of said memory areas **(graph of pointer relationships)**; determining a correlation between reference values corresponding to said at least one data item roots and memory areas allocated during said execution up

to said execution point **(graph of pointer relationships)** by identifying at least one data item reachable from said at least one data item roots; and performing a memory management operation on allocated memory areas in dependence upon said correlation. **(all citations from Wilson: Page 9, Section 2.2, Paragraph 1)**

4. **Regarding Claim 2, 12, and 22,** Wilson discloses wherein each of said at least one data items is an operand. **(Wilson: Page 5, Section 1.3) The paper makes the simplification that objects being collected are from the variety of types possible and that it is easy to determine the type of an object.**

5. **Regarding Claim 3, 13, and 23,** Wilson discloses wherein said identifying step comprises: identifying a possible execution path leading to said execution point, wherein said possible execution path may be different from said actual execution path; performing a simulated execution of said possible execution path; and wherein said at least one data item roots and said at least one data items accessible to said processing task are identified by following said possible execution path to said current execution point. **Traversing the graph of pointer relationships, usually by either depth-first or breadth-first traversal. (Wilson: Page 9, Section 2.2, Paragraph 1)**

6. **Regarding Claim 4, 14, and 24,** Wilson discloses wherein said memory management operation comprises marking all of said memory areas that are accessible to said processing task either directly or indirectly through said identified data items **(The objects that are reached are marked in some way; Wilson: Page 9, Section 2.2, Paragraph 1.)** and collecting unmarked memory areas for re-allocation during subsequent execution of said processing task. **(memory is swept ... find all**

unmarked objects and reclaim their space; Wilson: Page 9, Section 2.2, Paragraph 2)

7. **Regarding Claim 5, 15, and 25,** Wilson discloses wherein said memory management operation comprises compacting said unmarked memory areas prior to reallocation. **(Wilson: Page 10, Section 2.3, Lines 1-7)**

8. **Claims 6, 16, and 26 are rejected using the same rationale as claim 3.**

9. **Regarding Claim 9, 19, and 29,** Wilson discloses wherein said processing task is a component of a computer program written in an object-oriented programming language. **(Wilson: Page 2, Section 1, Lines 23-28)**

Claim Rejections - 35 USC § 103

10. The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all obviousness rejections set forth in this Office action:

> (a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.

11. Claims 10, 20, and 30 are rejected under 35 U.S.C. 103(a) as being unpatentable over Wilson ("Uniprocessor Garbage Collection Techniques").

12. **Regarding Claim 10, 20, and 30,** Wilson fails to disclose wherein said object oriented programming language is Java.

13. However, the Office takes Official Notice that it would have been obvious to a person of ordinary skill in the art to use Java as the object oriented programming language of Wilson.

14. The motivation for doing so would have been that Java allows the same program to be run on multiple operating systems, it contains built in support for networks, and it is a well understood and commonly accepted programming language among programmers.

15. Therefore, it would have been obvious to use Java as the object oriented programming language in Wilson for the benefits of allowing the program to run on multiple operating systems, containing built in support for networks, and being well understood and commonly accepted among programmers to obtain the invention as specified in claims 10, 20, and 30.

16. Claims 7, 8, 17, 18, 27, 28, and 31-33 are rejected under 35 U.S.C. 103(a) as being unpatentable over Wilson () as applied to claim 6 above, and further in view of Hosoya et al. ("Garbage Collection via Dynamic Type Inference" (herein after Hosoya)).

17. **Regarding Claims 7, 17, 27, and 31-33,** Wilson discloses scanning a plurality of program instructions corresponding to said programming task and logging a data type for each store instruction corresponding to each of said at least one data items; and simulating all possible execution paths up to said execution point for each of said at least one data item root or said at least one data items. **Traversing the graph of**

pointer relationships, usually by either depth-first or breadth-first traversal. (Wilson: Page 9, Section 2.2, Paragraph 1)

Wilson fails to disclose categorizing at least one of said data item roots or said at least one data items as a multiple-type variable if different data types are logged for different store instructions for a respective data item; determining the data type associated with each multiple-type variable at each of said plurality of program instructions for each of said possible execution paths; and checking said determined data type for each of said multiple-type variables at one of said plurality of program instructions corresponding to said current execution point; and said memory management operation is performed in dependence upon a result of said step of checking said determined data type.

Hosoya discloses categorizing at least one of said one or more data items as a multiple-type variable if different data types are logged for different store instructions for a respective data item; determining the data type associated with each multiple-type variable at each of said plurality of program instructions for each of said possible execution paths; and checking said determined data type for each of said multiple-type variables at one of said plurality of program instructions corresponding to said current execution point; and said memory management operation is performed in dependence upon a result of said step of checking said determined data type. **(Hosoya: Abstract, Lines 2-6 and Page 216, Lines 3-11)**

Wilson and Hosoya are analogous art because they are from the same field of endeavor, garbage collection.

At the time of the invention it would have been obvious to a person of ordinary skill in the art to combine the type inference garbage collection of Hosoya into the garbage collection techniques of Wilson.

The motivation for doing so would have been that the garbage collection technique of Hosoya collects more garbage than any other algorithm using the same type system does. **(Hosoya: Page 233, Section 9, Lines 4-7)**

Therefore, it would have been obvious to combine the type inference garbage collection of Hosoya into the garbage collection techniques of Wilson for the benefit of collecting more garbage than any other algorithm using the same type system does to obtain the invention as specified in claims 17, 18, and 19.

18. **Regarding Claims 8, 18, and 28,** Wilson fails to disclose wherein said memory management operation involves tagging said at least one data item as suitable for reallocation if said determined data type is different for different ones of said possible execution paths at said current execution point.

Hosoya discloses wherein said memory management operation involves tagging said data item as suitable for reallocation if said determined data type is different for different ones of said possible execution paths at said current execution point. **(Hosoya: Abstract, Lines 2-6 and Section 1, Lines 1-5) Where Hosoya does not specifically say that semantic garbage will be reallocated it is understood that when speaking about garbage collection, garbage will be collected and reallocated.**

(10) Response to Argument

A. Regarding the lack of the suspension of an execution path, the applicant contends that Wilson teaches a mark-sweep technique that the Applicant discloses in the Applicant's specification. However, the only reference to mark and sweep techniques that occurs in the prior art section of the Applicant's specification merely states that the mark and sweep technique is a known algorithm for garbage collection. Also, just because the technique is mentioned in the Applicant's specification does not mean that the reference fails to teach the limitations disclosed in the Applicant's claims. This leads into the next point of contention.

The Applicant also contends that the Wilson reference discloses performing the mark-sweep technique prior to execution and therefore could not suspend an execution path. However, the mark-sweep technique as referenced in the Applicant's disclosure and as referenced in Wilson, is taught as an algorithm. (Wilson: Page 5, Section 1.4, Paragraphs 1-2.) Also, Wilson goes on, in that section, to state that incremental and generational schemes will be taught in sections 3 and 4, respectively. (Wilson: Page 5, Section 1.4, Paragraphs 3-4.) The significance of the statement that the mark-sweep technique is an algorithm means that it is a concept for implementing a larger garbage collection scheme. Therefore, the mark-sweep technique is not performed as the Applicant has alleged, and indeed is performed during the execution of the program.

Finally, the Applicant contends that the Examiner has improperly based an anticipation rejection on a combination of Wilson and an allegation of "known to one of skill in the art". However, the examiner has inartfully expressed the assertion, that garbage collection occurs at a point in an execution path where execution is suspended

in order to perform garbage collection, is inherent in the reference. In order to support the assertion of inherency the Examiner has provided the reference Gupta et al., herein after Gupta. In Figures 1, 2, and 3 on pages 4 and 5 Gupta shows execution paths and the "Stop-the-world pause" caused by garbage collection schemes. More importantly, Gupta shows that even concurrent mark-sweep collectors contain a pause in execution. Gupta describes, on Page 3, in the section titled "Concurrent mark-sweep (CMS) collector", that the concurrent collector is "mostly concurrent", and shows in the figures that even a collector which is entitled concurrent still has a pause, or suspension, in the execution path to perform garbage collection. Gupta shows that all of the collectors possess this pause.

B. Regarding the lack of an identification step, the Applicant contends that because Wilson operates prior to execution and fails to suspend execution that it cannot disclose an identification step. However, as has been shown above, Wilson operates during execution and does perform a suspension step, therefore Wilson does teach an identification step as previously stated in the rejection at Wilson: Page 9, Section 2.2, Paragraph 1, specifically referring to the graphing of pointer relationships.

C. Regarding the lack of a determining step, the Applicant contends that because Wilson fails to teach the suspension of execution and the identification step that Wilson cannot teach a determining step. However, as shown above, Wilson does suspend execution and teaches a determining step, therefore Wilson teaches a determining step as previously stated in the rejection at Wilson: Page 9, Section 2.2, Paragraph 1, specifically referring to the graphing of pointer relationships.

D. Regarding the lack of a memory management step, the Applicant contends that because Wilson fails to disclose the suspension step, the identification step, and the determining step that Wilson cannot disclose a memory management step. However, as shown above, Wilson does teach a suspension step, an identification step, and a determining step, therefore Wilson teaches a memory management step as previously stated in the rejection at Wilson: Page 9, Section 2.2, Paragraph 1.

E. Regarding the Wilson reference, the Applicant contends that the reference would lead one of ordinary skill in the art away from the Applicant's claimed invention. Applicant contends the Wilson teaches performing the mark-sweep technique of garbage collection prior to execution and therefore fails to disclose suspending the execution. However, the Applicant has failed to show any citations in Wilson that affirm this statement. Furthermore, the mark-sweep technique as referenced in the Applicant's disclosure and as referenced in Wilson, is taught as an algorithm. (Wilson: Page 5, Section 1.4, Paragraphs 1-2.) Also, Wilson goes on, in that section, to state that incremental and generational schemes will be taught in sections 3 and 4, respectively. (Wilson: Page 5, Section 1.4, Paragraphs 3-4.) The significance of the statement that the mark-sweep technique is an algorithm means that it is a concept for implementing a larger garbage collection scheme. Also, In Figures 1, 2, and 3 on pages 4 and 5 of Gupta, the reference shows execution paths and the "Stop-the-world pause" caused by garbage collection schemes. More importantly, Gupta shows that even concurrent mark-sweep collectors contain a pause in execution. Gupta describes, on Page 3, in the section titled "Concurrent mark-sweep (CMS) collector", that the concurrent collector is

"mostly concurrent", and shows in the figures that even a collector which is entitled concurrent still has a pause, or suspension, in the execution path to perform garbage collection. Gupta shows that all of the collectors possess this pause. Therefore, the mark-sweep technique of Wilson is not performed as the Applicant has alleged, and indeed is performed during the execution of the program. Therefore Wilson would not lead one of ordinary skill in the art away from the Applicant's claimed invention.

F. Regarding the rejection of claims 1-6, 9, 11-16, 19, 21-26, and 29 under 35 USC § 102, the Applicant contends there is no basis for rejection of the claims under 35 USC § 102 due to the arguments presented in Sections A-D. However, above the Examiner has traversed these arguments in their respective sections. Therefore, the rejections under 35 USC § 102 upheld.

G. Regarding the rejection of claims 10, 20, and 30, under 35 USC § 103, the Applicant contends that the rejections fail using the arguments presented in sections A-F. However, above the Examiner has traversed these arguments in their respective sections. Therefore, the rejections under 35 USC § 103 upheld.

H. Regarding the rejection of claims 7, 8, 17, 18, 27, 28, and 31-33 under 35 USC § 103, the Applicant contends that the rejections fail using the arguments presented in sections A-G. However, above the Examiner has traversed these arguments in their respective sections. Therefore, the rejections under 35 USC § 103 upheld.

(11) Related Proceeding(s) Appendix

No decision rendered by a court or the Board is identified by the examiner in the Related Appeals and Interferences section of this examiner's answer.

For the above reasons, it is believed that the rejections should be sustained.

Respectfully submitted,

Michael McFadden

8/3/07

Conferees:

Lynne Browne

~~Examiner~~ APPEAL PRACTICE SPECIALIST, TQAS
TECHNOLOGY CENTER 2100

Hyung Sough

HYUNG SOUGH
SUPERVISORY PATENT EXAMINER
8/06/07

Article

Turbo-charging Java HotSpot Virtual Machine, v1.4.x to Improve the Performance and Scalability of Application Servers

Print-friendly Version

Articles Index

By Alka Gupta and Michael Doyle
November 2002

This paper describes the requirements of a Telecommunication (Telco) application server written in the Java programming language, and how those requirements might influence Java virtual machine (JVM[1]) design. Most of the discussion is not limited to Telco alone, but applies to the enterprise as well. The paper:

- Gives an overview of two new garbage collection (GC) implementations, available in the Java HotSpot JVM starting with Java 2 Platform, Standard Edition (J2SE) 1.4.1 that are designed to make garbage collection less disruptive
- Describes the effects of garbage collection on Java applications' performance and scalability
- Covers other techniques for tuning garbage collection
- Includes some more tips and tactics for analysis and performance improvement of Java applications
- Provides two tools, PrintGCStats and ThreadAnalyser, for download, to help with garbage collection and JVM thread performance analysis.

Finally, the paper includes a case study of a Session Initiation Protocol (SIP) server that demonstrates significant performance gains from using these new collectors and other tuning techniques described.

The Java programming language is widely accepted as the language of choice for many data-oriented applications. The reasons for the overwhelming success of the language are based around a key set of features that were designed into the language from its inception:

- WORA -- Write Once Run Anywhere
- Automatic memory allocation and collection
- Object Oriented Design
- Inheritance

But crucially, the design of the underlying JVM, which is the engine that executes the Java byte code produced during compilation of source code, has been geared towards general data applications. This has led to some anomalies in run-time performance and responsiveness when using the JVM in a time-critical environment. Traditionally, GC suspends the user's application while the system re-cycles memory. Such GC pauses can be of the order of seconds or even minutes in severe cases.

The Telecom Carriers equipment suppliers are keenly aware that carriers are placed under stringent Service Level Agreements with corporate customers, as well as having an expected level of reliability and response time to the residential market. The idea of waiting 10 or 20 seconds to hear the dial-tone when you pick up the phone is clearly unacceptable.

Telecom Carriers are interested in using Java within their environment for a variety of reasons, not least of which is the vast wealth of application creation talent that exists -- literally millions of programmers and companies have standardized on Java as the base technology, which means that new services and skills are attainable in quantities previously unheard of. Add to that the comprehensive and standard environment of Java 2 Platform, Enterprise Edition (J2EE), and the possibilities expand exponentially. However, the specter of "pauses" and the resulting latency in application response would cause any carrier to question the validity of Java as the core technology within their networks.

Carriers depend on completed calls and service sessions in order to generate revenue. Incomplete or

abandoned sessions, (for example, calls and Instant Messages), represent wasted network resources, and a cost of provisioning bandwidth that is ultimately unused.

SIP as a protocol is an ideal choice for this environment, as it is an accepted standard within the Carrier market, including the 3rd generation (3G) companies spearheading the next generation communication infrastructure. SIP is a UDP and TCP-based signaling protocol that has packet retransmission (for UDP) built into the protocol definition. If a UDP request packet does not have a response within a certain time frame (about 500 milliseconds (ms) usually), then retransmission occurs, to account for network outage, packet loss and other errors that must be addressed. This means that the JVM cannot suspend an application for any period even approaching 500 ms, because when network packet latency (50-100ms each way), and processing time on the CPU are taken into account, there is very little time left before a retransmission would occur. Cascade failure is often a result of poorly designed application servers that do not accommodate high load conditions, or run on JVMs (such as those prior to J2SE1.4) that use traditional single threaded, stop-the-world garbage collection policies, meaning they pause or suspend all application threads during garbage collection. Also, these JVMs are single-threaded and enable garbage collection only on a single CPU even within a multi-CPU host.

Sun HotSpot JVMs (JDK 1.3+) are generational. The generational algorithm provides for efficient memory recycling and object aging. The JVM heap is split into a "young generation" and an "old generation" according to object "age". The young generation is further split into an "Eden" and two "Survivor spaces". For most applications, two-thirds of allocated objects die very young, are considered "short term objects", and can be collected in the young generation. Typically, the young generation is much smaller in size relative to the total heap size. This leads to frequent but short pauses in the young generation, but more memory is recovered per unit of collection work. However, objects that survive a sufficiently large number of young generation collections are considered "old" or "long term objects" and are "promoted" or "tenured" to the old generation. Even though the old generation is typically larger, it eventually gets filled up and requires collection. This leads to less frequent but larger pauses in the old generation. For further information on Sun HotSpot JVM garbage collection technology, refer to Tuning Garbage Collection with the 1.3.1 Java Virtual Machine.

Ubiquity's SIP application server, Application Services Broker (ASB), is 100% pure Java technology, and provides a perfect testing environment for the JVM and garbage collection. As a SIP Application server, the ASB is responsible for receiving, modifying and ultimately responding to service requests from the signaling network. It achieves this by directing the SIP traffic from the signaling network to service elements (SIP Servlets) that are running within the ASB. These service elements register with the ASB for interest in certain signaling messages. The ASB generates large call volumes with a variety of load distribution characteristics. It is also designed as a cluster-based implementation, scaling on multiple processors, on a single machine or multiple hosts. This is where the JVM design becomes stressed to the limits, and observation of this environment has provided considerable insights for improving the multithreaded nature of the JVMs.

The Ubiquity ASB for example, generates approximately 220 KB of garbage per call processed (in a general sense, a call is a session). Almost 10% data of each call must survive for about 40 seconds, and the other 90% dies almost immediately. This 10% of the data, even though short lived, lives long enough to consume enormous amounts of memory if one considers an average performance of 100 calls per second. In 40 seconds this would account for at least 88 MB of active data. If garbage collection were to happen, for example, once every five minutes in the old generation, for example, the heap would need to be sized to at least 660 MB. That is a huge amount for the traditional garbage collection engine to scan and collect within, say, 100-150 ms.

Telco Application servers require a more deterministic GC model that is restricted in the time it suspends applications, and also that scales well on multiple processors. In J2SE 1.2 and 1.3, the GC is single-threaded and stop-the-world in nature. The pauses resulting from garbage collection in the JVM add latency to the application and adversely affect the application's performance in terms of throughput and scalability. This architecture also precludes any significant scaling within a multi-CPU single box machine. Since the garbage collection is not distributed across the many available processors, with a highly multi-threaded application running on the JVM, the JVM becomes stressed in a multi-CPU environment, in that the application will be constantly waiting for the JVM to return control, even though there are sufficient processors available to perform application-level work. The impact of a single-threaded GC on a multiprocessor system grows relative to an otherwise parallel application. If we assume an ideal application is perfectly scalable with the exception of GC, it may spend only 1% of the time in GC on a uniprocessor. This translates into roughly 24% loss in throughput at 32 processors. If the GC time for a uniprocessor is as much as 10%, this is still not considered an outrageous amount of time in GC. However, when scaling up, this translates to roughly 78% of lost throughput which is extremely significant.

Turbo-charging Java in this context would essentially mean the following:

1. Utilize machines with larger numbers of CPUs
2. Access much larger memory spaces
3. Handle more concurrent socket connections

J2SE1.4 implements the non-blocking new I/O APIs which provide new features and improved performance in the areas of buffer management, scalable network and file I/O.

- The new network I/O package dramatically increases the number of simultaneous connections that a server can handle by removing the need to dedicate one thread to every open connection.
- New file I/O supports read, write, copy, and transfer operations that are up to twice as fast as the current file I/O facilities. It also supports file locking, memory-mapped files, and multiple concurrent read/write operations.

The 64-bit JVM in J2SE1.4 allows heaps larger than 4G, even up to 300G or more. However, a bigger heap results in bigger, though less frequent sequential GC pauses, if garbage generation and allocation rates are held constant.

An important parameter that determines the application efficiency and scalability is the "GC sequential overhead". It is the percentage of the application run time spent doing garbage collection in the JVM, while the application is paused. It can be calculated as:

```
Avg. GC pause * Avg. GC frequency * 100 %.
```

"GC frequency" is the periodicity of GCs or number of GCs per unit time. GC sequential overhead may be calculated separately for the "young" and the "old generation" since the two generations have quite different average GC pause and frequency characteristics, and may be added to calculate the total GC sequential overhead for the application.

GC sequential overhead can also be calculated as

```
Total GC time/Total wall clock run time.
```

"Total GC time" can be calculated as

```
Avg. GC pause * total no. of GCs
```

Clearly, smaller GC sequential overhead implies higher application throughput and scalability.

The New Garbage Collectors

J2SE1.4.1 introduces two new garbage collectors designed to help boost application performance and scalability by allowing the JVM to scale to larger number of CPUs and memory. These collectors help meet some of the challenges imposed by Telco requirements such as:

- GC sequential overhead on a system may not be more than 10%, to ensure scalability and optimal use of system resources for maximum throughput
- Any single GC pause during the entire application run may be no more than 200 ms, to meet the latency requirements as set by the protocol between the client and the server, and to ensure good response times by the server.

These two new collectors introduced in J2SE1.4.1 are:

- Parallel collector
  The parallel collector is implemented in the young generation. It is multi-threaded and stop-the-world. This collector enables garbage collection to occur on multiple threads for better performance on multiprocessor machines. Even though it suspends all "mutators" (application threads), it is able to complete the given amount of garbage collection work much more quickly by leveraging all available CPUs on the system. This reduces the GC pauses in the young generation significantly. The parallel collector thus enables the applications to scale to larger number of CPUs as well as larger memory.

- Concurrent mark-sweep (CMS) collector
  The concurrent mark-sweep collector (CMS) is implemented in the old generation. The CMS collector executes "mostly concurrently" with the application, hence, is sometimes referred to as the "mostly-concurrent garbage collector". It trades the utilization of processing power that would otherwise be available to the application for shorter garbage collection pause times. The CMS collection is split into four phases:
  - Initial mark
  - Concurrent marking
  - Remark
  - Concurrent sweeping

  The "initial mark" and and "remark" phases are stop-the-world phases, in which the CMS collector

has to "suspend" the mutators. During initial mark, it records all objects directly reachable from the "roots" of the system. During the "concurrent marking" phase, mutators are resumed and a concurrent marking phase is initiated. During the "remark" phase, the mutators are once again suspended to complete the final marking, and finally, during the "concurrent sweeping" phase, mutators are resumed and all unmarked objects are deallocated, while concurrently sweeping over the heap. The initial mark and remark pauses are quite minimal. For an old generation size of 1G, they may be ~ 200 ms or less. The concurrent sweeping phase, when the garbage is collected, may still take as much time as the mark-compact collector; however, this pause is hidden as the mutators are not suspended. The "mostly concurrent" nature of the CMS collector enables the JVM to scale to larger heaps and CPUs, thus addressing both the latency and throughput issues arising from the default mark-compact stop-the-world collector.

Table 1 highlights the features of the different collectors available in J2SE1.4.1.

Young Generation Collectors	Old Generation Collectors
Copying collector	**Mark-compact collector**
Default Stop-the-world Single threaded All J2SEs	Default Stop-the-world Single threaded All J2SEs
Parallel collector	**Concurrent mark-sweep collector**
Stop-the-world Multi-threaded J2SE1.4.1+	Mostly-concurrent Single threaded J2SE1.4.1+

Table 1. Summary of the features of different garbage collectors

Figures 1, 2 and 3 are graphical illustrations of the different garbage collectors. The green arrows represent a multi-threaded application running on a multi-CPU box. The red arrows represent the GC threads. The length of the GC thread roughly represents the length of the GC pause.

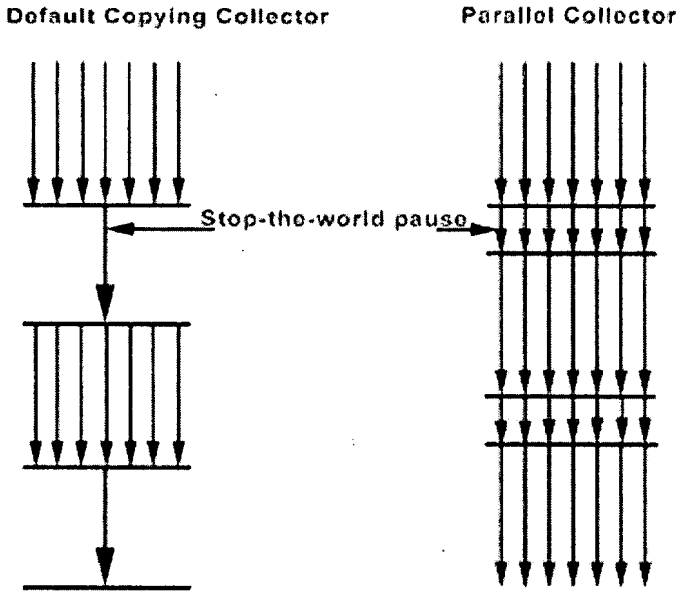

Figure 1. Young generation collectors.

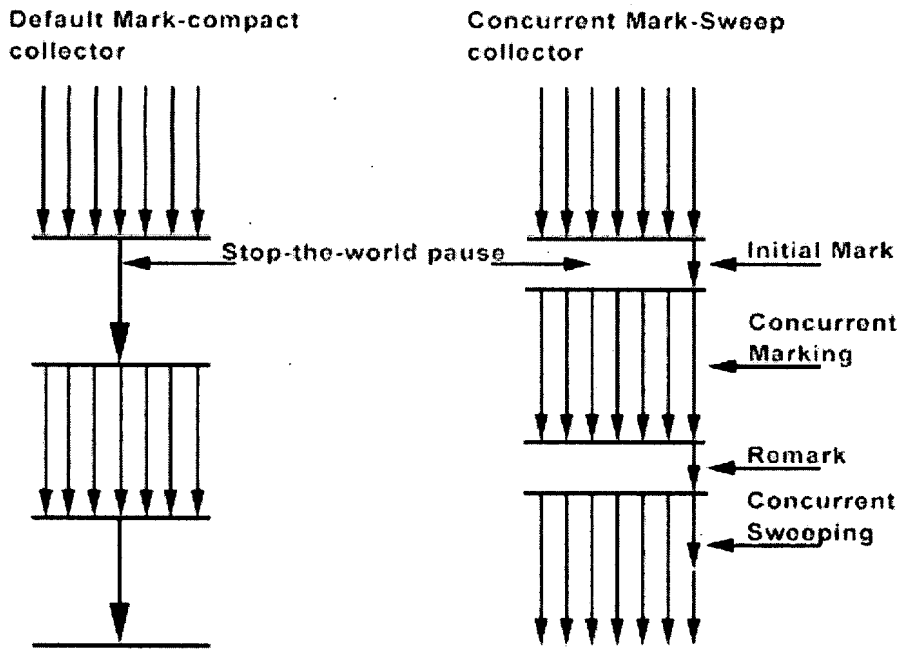

Figure 2. Old generation collectors. Garbage collection with the Parallel Collector and Concurrent mark-sweep collector enabled together. Best Case scenario for high throughput and low latency.

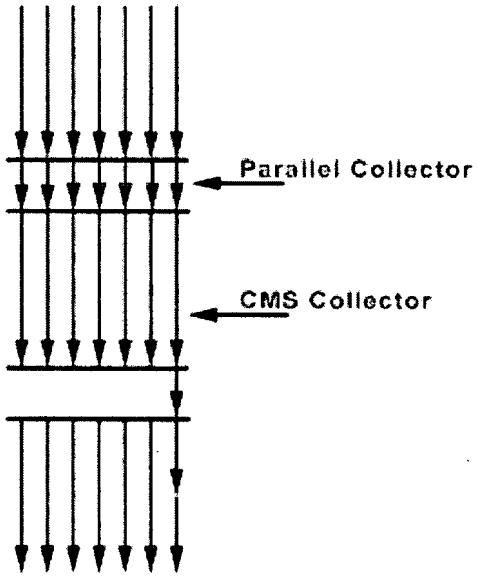

Figure 3. Garbage Collection with the two new collectors enabled together.

As illustrated by the diagrams above, use of the parallel collector in the young generation and the concurrent mark-sweep collector in the old generation together can help reduce pause times and GC sequential overhead. These two collectors thus help the applications to scale to larger number of processors, and to larger memory.

JVM Switches and Options for Tuning the JVM from a GC Perspective

Some of the well known switches which have been available in Sun HotSpot JVMs from J2SE1.3 for the purpose of sizing the heap and tuning garbage collection include:

- **General Switches**
  - `-server`
  - `-Xmx, -Xms`
  - `-XX:NewSize=, -XX:MaxNewSize=`
  - `-XX:SurvivorRatio=`

For more details on these switches, refer to JVM HotSpot VM options.

The switches available from J2SE1.4.1 for enabling the two new garbage collectors are:

- **Parallel Collector**
  - `-XX:+UseParNewGC`
    This flag turns on parallel garbage collection in the young generation. It can be enabled together with the CMS collector in the old generation. Available in J2SE1.4.1 on an experimental basis.

  - `-XX:ParallelGCThreads=n`
    This switch sets the number of parallel GC threads that the JVM must run for performing garbage collection in the young generation. The default value of `n` is equal to the number of CPUs on the system. However, it has been observed that tweaking this number may improve performance in some cases. An example might be that of multiple instances of JVM running on a single multi-CPU system. In this case, the number of parallel GC threads for each JVM may need to be set to less than the number of CPUs by using this switch. Available from J2SE1.4.1.

  - `-XX:+UseParallelGC`
    This flag also turns on parallel garbage collection policy in the young generation; however, it does not work with the CMS collector in the old generation. It is more suitable for enterprise applications which can benefit from very large young generation heaps.

- **Concurrent Collector**
  - `-XX:+UseConcMarkSweepGC`
    This flag turns on concurrent garbage collection in the old generation. Available from J2SE1.4.1.

  - `-XX:CMSInitiatingOccupancyFraction=x`
    Sets the threshold percentage of the used heap in the old generation at which the CMS collection takes place. For example, if set to 60, the CMS collector will be initiated every time the old generation becomes 60% full. By default, this threshold is calculated at run time, and the CMS collector might be triggered only when the old generation heap is about 80-90% full. Tuning this value can improve performance in many cases. Since the CMS collector does not suspend the mutators as it sweeps and frees memory, setting this switch can ensure that enough free memory is available for object promotion from the young generation as more data is allocated by the application. Sometimes, if this switch is not tuned, CMS collection might not be able to keep up and may fail, eventually triggering the default stop-the-world mark-compact collector. Available from J2SE1.4.1.

Some of the other switches which can be used for performance tuning are as follows:

- `-XX:MaxTenuringThreshold=y`
  This switch determines how much the objects may age in the young generation before getting promoted to the older generation. The default value is 31. For a big enough young generation and "survivor space", the long-lived objects may be copied up to 31 times between the survivor spaces before they are finally promoted to the old generation. For most Telco applications, it has been found that 80-90% of the objects that are created per call or session die almost immediately after they are created, and the rest (10-20%) survive the entire duration of that call. Setting `-XX:MaxTenuringThreshold=0` promotes all objects allocated by the application in the young generation, which survive a single GC cycle, directly to the old generation without copying them around between the survivor spaces in the young generation. This setting, when used with CMS collector in the old generation helps in two ways.
  - The young generation GC does not have to waste its time copying the 10-20% long lived objects multiple times between the survivor spaces, only to finally promote them to the old generation.
  - Additionally, most of the collection and cleaning work of these objects can be done "concurrently" in the old generation. This behavior leads to additional reduction in GC sequential overhead.

  When this switch is used, it is advisable to set the `-XX:SurvivorRatio` to a very high value, say 128. This is because, in this case, the survivor spaces are not used, and objects are promoted directly from Eden to the old generation during every GC cycle. By setting a high survivor ratio, most of the the young generation heap is allocated to "Eden". Available from J2SE1.3.

- `-XX:TargetSurvivorRatio=z`
  This flag sets the desired percentage of the survivor space heap which must be used before objects are promoted to the old generation. For example, setting `z` to 90 would mean that 90% of the survivor space must be used before the young generation is considered full and objects are promoted to the old generation. This would allow objects to age more in the young generation before being tenured. The default value is 50. Available from J2SE1.3.

Mining the "verbose:gc" Log Files for Analysis and Performance Tuning the GC

There is a lot of useful GC-related information which the JVM can log into a file. This information can be used to tune and size the application and the JVM from a GC perspective. The switches available for

logging this information are:

- `verbose:gc`
  This flag turns on the logging of GC information. Available from J2SE1.3.
- `-Xloggc=filename`
  This switch can be used to specify the name of the log file where the "verbose:gc" information can be logged instead of standard output. Available from J2SE1.4.
- `-XX:+PrintGCTimeStamps`
  Prints the times at which the GCs happen relative to the start of the application. Available from J2SE1.4.
- `-XX:+PrintGCDetails`
  Gives some details about the GCs, such as size of the young and old generation before and after GCs, size of total heap, time it takes for a GC to happen in young and old generation, size of objects promoted at every GC etc. Available from J2SE1.4.
- `-XX:+PrintTenuringDistribution`
  Gives the aging distribution of the allocated objects in the young generation. Tuning of `-XX:NewSize`, `-XX:MaxNewSize`, `-XX:SurvivorRatio` and `-XX:MaxTenuringThreshold=0` as described earlier, should be directed by the analysis of the output from this switch to determine that objects are not prematurely promoted to the old generation. Available from J2SE1.3.

Snapshots from a "verbose:gc" Log File Generated by Using the Following Switches

```
java -verbose:gc -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -
XX:+PrintTenuringDistribution -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -
Xmx512m -Xms512m -XX:NewSize=24m -XX:MaxNewSize=24m -XX:SurvivorRatio=2 <app-
name>
```

Young Generation GC

```
311.649: [GC 311.65: [ParNew Desired survivor size 4194304 bytes, new
threshold 3 (max 31)
- age 1: 1848472 bytes, 1848472 total
- age 2: 1796200 bytes, 3644672 total
- age 3: 1795664 bytes, 5440336 total
: 21647K->5312K(24576K), 0.1333032 secs] 377334K->362736K(516096K), 0.1334940
secs]
```

The GC snapshot above describes a young generation GC and gives the following information:

- Total JVM heap at this GC was `516096K` and total heap size of the young generation was `24576K`.
- The old generation heap at the time of this GC was `516096K - 24576K = 491520K`.
- Describes the "aging distribution" of the objects allocated by the application in the young generation at the end of this GC cycle. This GC cycle set the `new threshold=3` meaning that objects will age up to 3 before the next GC cycle takes place.
- The *Desired survivor size* of 4194304 bytes was determined by `SurvivorRatio=2`, which sets the ratio of `Eden/Survivor space=2`. With a young generation heap set to 24MB, this would mean `24MB = 2 * Survivor space + Survivor space + Survivor space`, or `Survivor space=8MB`. With a default value of `TargetSurvivorRatio=50`, desired survivor size was set to 8 * .5 = 4MB
- Time stamp of the GC relative to the start of the application run was `311.649 seconds`
- Total GC pause for this collection in the young generation was `0.1334940 seconds`.
- The size of the used young generation heap before this GC was `21647K`
- The size of the used young generation heap after this GC was `5312K`
- `377334K - 362736K = 14598K` of data was garbage collected in the young generation.
- `21647K - 5312K - 14598K = 1737K` was promoted into the old generation.
- The total size of the used heap was `377334K` before this GC took place.
- The total size of the used heap was `362736K` after this GC took place.

- The size of used heap in old generation before this GC was `377334K - 21647K = 355687K`
- The size of used heap in old generation after this GC was `362736K - 5312K = 357424K`
- The total size of objects promoted into the old generation during this GC was `357424K - 355687K = 1737K`
- The legend `ParNew` indicates that the parallel collector was used in the young generation. The use of default copying collector is indicated by a legend `DefNew`, instead of `ParNew`.

The convention used in the verbose:gc log format is that if there's something being reported that is specific to a generation, it's preceded by the name of that generation.

`[GC [gen1 info1] [gen2 info2] info]`
`info` is general to GC and includes information for the young and old generation combined, `info<n>` is specific to `gen<n>`. So, attention needs to be paid to the nesting and matching of square brackets.

Old Generation GC

CMS Collector

The snapshots below indicate the use of the concurrent mark-sweep collector in the old generation. As mentioned earlier in the paper, this phase is split into four phases.

```
513.474: [GC [1 CMS-initial-mark: 335432K(491520K)] 340897K(516096K),
0.0482491secs]
```

- This is the `initial mark` phase which is stop-the-world and took 0.048 seconds.
- The time relative to the start of the application was 513.474 seconds.
- 335432K was the size of the old generation used heap.
- 491520K was the size of the total used heap including the young generation used heap.
- 340897K was the total size of the old generation heap.
- 516096K was the total size of the heap including the young generation.
- This phase does not recycle any memory.
- The prefix legend `" [GC"` represents the stop-the-world phase.

```
513.523: [CMS-concurrent-mark-start]
514.337: [CMS-concurrent-mark: 0.814/0.814 secs]
514.337: [CMS-concurrent-preclean-start]
514.36: [CMS-concurrent-preclean: 0.023/0.023 secs]
```

This is the concurrent mark phase which took less than a second, `(0.814 + 0.023) seconds`, however, the application runs concurrently with the GC during this time. Again, this phase does not collect any garbage.

```
514.361: [GC 514.361: [dirty card accumulation, 0.0072366 secs]
514.368: [dirty card rescan, 0.0037990 secs]
514.372: [remark from roots, 0.1471209 secs]
514.519: [weak refs processing, 0.0043200 secs] [1 CMS-remark: 335432K
(491520K)] 352841K(516096K), 0.1629795 secs]
```

- This is the stop-the-world `remark` phase which took about 0.162 secs.
- 335432K was the size of used heap in the old generation.
- 491520K was the total heap size of the old generation.
- 352841K was the total size of the used heap including the young generation.
- 516096K was the total size of the heap including the young generation.
- No memory is recycled during this phase.
- The legend `" [GC"`, as mentioned earlier, represents the stop-the-world phase.

```
514.525: [CMS-concurrent-sweep-start]
517.692: [CMS-concurrent-sweep: 2.905/3.167 secs]
517.693: [CMS-concurrent-reset-start]
517.766: [CMS-concurrent-reset: 0.073/0.073 secs]
```

This is the concurrent sweep phase which took about 3 seconds, however, application threads can run concurrently with the GC thread during this phase on a multi-processor system. Among the four CMS phases, this is the only phase when the heap is swept and collected.

Default Mark-Compact Collector

If instead of the CMS collector, the default mark-compact collector is used in the old generation, the old GC snapshot would look as:

```
719.2: [GC 719.2: [DefNew: 20607K->20607K(24576K), 0.0000341 secs]719.2:
[Tenured: 471847K->92010K(491520K), 2.6654172 secs] 492454K->92010K(516096K),
2.6658030 secs]
```

- `719.2` seconds was the time relative to the start of the application at which this GC took place.
- The legend `DefNew` indicates the use of default copying collector in the young generation.
- The legend `"[GC"` represents a stop-the-world GC triggered by the JVM. For an old generation GC, requested by the application through the system call, `System.gc()`, the above snapshot would be prefixed by the legend `"Full GC"`
- The total heap size of the young generation was `24576K`.
- A young generation collection (only) was initially attempted. It was found that that collection couldn't be done because the old generation could not guarantee that it would absorb all potentially live data. As a result, only the young collection did not happen, and the young generation reported that no memory was reclaimed (notice that it all finished rather quickly, in a fraction of a millisecond). That's because nothing much happened beyond determining that the young generation collection couldn't happen:
  `[DefNew: 20607K->20607K(24576K), 0.0000341 secs]`

It was then decided that a "tenured" generation collection needed to happen, (obviously), because that generation was too full to absorb promotions from the young generation, so a full mark-compact collection was done.

- The legend `Tenured` indicates a full mark-compact GC in the old generation. The "old generation" is sometimes referred to as "Tenured generation"
- `471847K` was the size of used heap in the old generation before GC.
- `92010K` was the size of used heap in the old generation after the GC.
- `491520K` was the total heap size of the old generation at the time of this GC.
- `492454K` was the total used heap, for young and old generations combined, before GC.
- `92010K` was the total used heap, for young and old generations combined, after GC.
- Total garbage collected during this GC was `492454K - 92010K = 399837K`
- 516096K was the size of the total heap for the JVM.
- 2.6658030 seconds was the total time for which the application was suspended as a result of this GC.

Application Modeling Based on Data Mined from the verbose:gc Logs

A variety of information regarding the application and JVM behavior with respect to garbage collection can be derived from these logs. This includes:

- `Average GC pauses` in the young and old generation

  The average time the application is suspended while garbage collection is done in the JVM.

- `Average GC frequency` in the young and old generation

  The periodicity at which the garbage collector runs in the young and the old generation. This can be obtained since the time instance of each GC activity is logged.

- `GC sequential overhead`

  The percentage of system time for which the application is suspended for garbage collection to take place. Calculated as `Avg. GC pause * Avg. GC frequency * 100%`

- `GC concurrent overhead`

  The percentage of system time for which garbage collection happens concurrently with the application. Calculated as `Avg. concurrent GC time (sweeping phase) * Avg. concurrent GC frequency / no. of CPUs`

- Memory recycled by each GC in the young and old generation

  The total garbage collected during each GC.

- `Allocation rate`

  The rate at which data gets allocated by the application in the young generation. If young generation heap `occupancy_at_start_of_current_gc=x , occupancy_at_end_of_previous_gc = y` and the `GC frequency` is `1 per second`, then the `allocation rate` is approximately `x-y per second.`

- `Promotion rate`

  The rate at which data gets promoted to the old generation. Size of objects promoted per GC is calculated as described in the "Young Generation GC" section, and if the young generation `GC frequency` is 1 per second for example, the promotion rate would be 1737K per second for the snapshot shown in that section.

- `Total data` allocated by the application per call

  This can be calculated as `Allocation Rate /Call rate` where allocation rate is calculated as above and `Call Rate` is the load on the application server. In other words, Call rate is the rate at which the server is processing the incoming calls or requests.

- Total Data can be split into `short term data` and `long term data`

  `Long term data` is what survives the young generation GC cycles and is promoted to the old generation. This can be calculated as `Promotion Rate/Call Rate`

- `Short Term data` per call

  The short lived data that dies very quickly and is collected in the young generation. This can be calculated as `Total Data - Long Term Data.`

- Total active data per call

  This is critical for building a model for sizing the JVM heap. For example, for a load of 100 calls per second, with long term data of 50 K per call lasting for a minimum of 40 seconds, as in the case of SIP applications, the minimum memory footprint of the old generation would have to be `50K*40s*100 = 200M`

- "Memory leaks"

  These can be detected and "out of memory" errors can be better understood by monitoring the garbage collected at each GC as shown by these logs.

This kind of information can be used to model and better understand the application and JVM behavior from a GC perspective, then to performance tune the process of garbage collection.

PrintGCStats: Tool for Mining the "verbose:gc" Logs for the Purpose of Analyzing and Tuning Garbage Collection

PrintGCStats is a shell script which mines the "verbose:gc" logs and summarizes statistics about garbage collection, in particular, the GC pause times (total, averages, maximum and standard deviations) in the young and old generations. It also calculates other important GC parameters like the GC sequential overhead, GC concurrent overhead, data allocation and promotion rates, total GC and application time and so on. In addition to summary statistics, PrintGCStats also provides the timeline analysis of GC over the application run time, by sampling the data at user specified intervals.

- **Input:**

  The input to this script should be the output from the HotSpot Virtual Machine when run with one or more of the following flags.

- `-verbose:gc`
  Produces minimal output, so statistics are limited, but available in all JVMs

- `-XX:+PrintGCTimeStamps`
  Enables time-based statistics (for example, allocation rates, intervals), but only available from J2SE 1.4.0.

- `-XX:+PrintGCDetails`
  Enables more detailed statistics gathering, but only available from J2SE 1.4.0.

- Recommended command-line with J2SE 1.4.1 and later:
  `java -verbose:gc -XX:+PrintGCTimeStamps -XX:+PrintGCDetails ...`

- **Usage:**

```
PrintGCStats -v ncpu=<n> [-v interval=<seconds>] [-v verbose=1]
<gc_log_file >
```

  - `ncpu`
    Number of cpus on the machine where the Java application was run. Used to compute cpu time available and GC 'load' factors. No default; must be specified on the command line (defaulting to 1 is too error prone).

  - `interval`
    Print statistics at the end of each interval to provide timeline analysis; requires output from -XX:+PrintGCTimeStamps. Default is 0 (disabled).

  - `verbose`
    If non-zero, print each item on a separate line in addition to the summary statistics.

- **Output statistics:**

  Table 2 describes the output statistics from PrintGCStats.

Item name	Description
gen0(s)	Young generation collection time in seconds
cmsIM(s)	CMS initial mark pause in seconds
cmsRM(s)	CMS remark pause in seconds
GC(s)	All stop-the-world GC pauses in seconds
cmsCM(s)	CMS concurrent mark phase in seconds
cmsCS(s)	CMS concurrent sweep phase in seconds
alloc(MB)	Object allocation in young generation in MB
promo(MB)	Object promotion to old generation in MB
elapsed_time(s)	Total wall clock elapsed time for the application run in seconds
tot_cpu_time(s)	Total CPU time = no. of CPUs * elapsed_time
mut_cpu_time(s)	Total time that was available to the application in seconds
gc0_time(s)	Total time used by GC during young generation pauses
alloc/elapsed_time(MB/s)	Allocation rate per unit of elapsed time in MB/seconds
alloc/tot_cpu_time(MB/s)	Allocation rate per unit of total CPU time in MB/seconds
alloc/mut_cpu_time(MB/s)	Allocation rate per unit of total application time in MB/seconds
promo/gc0_time(MB/s)	Promotion rate per unit of GC time in MB/seconds
gc_seq_load(%)	Percentage of total time spent in stop-the-world GCs
gc_conc_load(%)	Percentage of total time spent in concurrent GCs
gc_tot_load(%)	Total percentage of GC time (sequential and concurrent)

Table 2. Summary of the output statistics from PrintGCStats

- Download PrintGCStats

Other Tips and Tricks from the Trenches for Performance Analysis and Improvement of Java Servers on the Solaris Platform

These tuning techniques recommended below are based on the author's experience in the field.

- Use the `alternate thread library` available from Solaris 8 onwards. It can be used by setting the `LD_LIBRARY_PATH=/usr/lib/lwp:/usr/lib` on Solaris 8 and is the default on Solaris 9. It invokes the "one-to-one threading model" between the Java threads and the kernel threads. 5-10% or more performance improvement in throughput has been observed in many cases with the use of this library.

- Use `prstat -Lm -p <jvm process id>` to analyze the resource usage of a process on a per light-weight-process (LWP) basis and to identify potential bottlenecks in terms of scalability and performance. It helps to see, for example, if GC is the bottleneck with respect to application scalability. In many cases, the application itself is not well threaded, hence does not scale on larger systems. This command reports system resource usage and activity on a per `LWP` basis within a given process. Run `man prstat` on a command line for more details. One of the drawbacks of this command output is that it is difficult to identify which "Java thread" might be associated with the reported `LWP id`.

- **`ThreadAnalyser`** is a shell script to analyze a Java process and produce "thread names" in the `prstat` output, instead of Solaris `LWPIDs`. It helps identifying the system resource usage on a per Java application's "thread name" basis. The most ideal way to use the script is with the `alternate thread library` as mentioned above. ThreadAnalyser can attach to an already running Java process (as long as it has access to the stderr output from the process, having previously been redirected to a file) or it can start the Java process itself, given a startup script file for the Java process.

Example usage:
Start a Java process and redirect the standard error to a file as in "`java <app_name> >/tmp/java.out 2>&1`". Also note the `PID` of the Java process. Then, attach the ThreadAnalyser as this:

```
ThreadAnalyser -f /tmp/java.out -p PID <interval>
```

or

```
ThreadAnalyser -f /tmp/java.out <interval>
```

if there is no other Java process running on this machine (it finds the PID using ps(1))

To have ThreadAnalyser start the Java process, run as

```
ThreadAnalyser -s <app_start_script_file> <interval>
```

This will work when there are no other Java processes running on the system.

`app_start_script_file` is the name of the startup script file that starts the Java application.
`interval` is the time interval at which output data from the script is updated. Default is 5.

For more details on other options and usage, read the README section of the `ThreadAnalyser` script.

Sample output format:

PID	USERNAME	USR	SYS	TRP	TFL	DFL	LCK	SLP	LAT	VCX	ICX	SCL	SIG	PROCESS/LWPID
12803	root	65	60	0.0	0.0	0.0	100	0.0	0.1	143	0	286	0	java/5:App-Thread-7
12803	root	25	20	0.0	0.0	0.0	100	0.0	0.0	10	0	30	0	java/6:STOPPER
12803	root	10	10	0.0	0.0	0.0	100	0.0	0.0	10	0	10	0	java/11:<Async-GC thread>
12803	root	5	10	0.0	0.0	0.0	0.0	100	0.0	0	0	0	0	java/15:App-Thread-9

(Click image to enlarge.)

Run `man prstat` on the system command line to get details on each of the columns in the above output.

Download ThreadAnalyser

- To get a full thread dump of a running Java application, send a `SIGQUIT` signal to the JVM process. This can be done from command line as `kill -QUIT <JVM process pid>`.

- To get a `hex` and symbolic stack trace for each LWP in the JVM process, run the command `pstack <JVM process id>`. The output also gives the mapping information of `LWP ids` to Java application thread `ids`. If the "alternate thread library" is used by the JVM, the output will show a one-to-one mapping between each of the process's LWP and the corresponding Java application thread bound to it. Run `man pstack` on command line for more details.

- Use the `-Xrunhprof` command line flag available in the JVM to help identify unnecessary object retention (sometimes imprecisely called "memory leaks").

- For UDP-based applications, minimize the number of times the `DatagramSocket.connect (InetAddress address, int port)` and `DatagramSocket.disconnect()` methods are called. As these are implemented as native calls in the JVM from version J2SE1.4 onwards, which provides other major benefits, incorrect usage of this API may account for a degradation in applications, specifically, improper use of this API can be very expensive in terms of an application's "system time".

A Real World Case Study Using the New Parallel and Concurrent Mark-Sweep Collectors

Described here are some of the results obtained using the two new garbage collection policies as well as the other tuning techniques detailed in this paper. For the purpose of this case study, a SIP Application Server from Ubiquity, Application Services Broker (ASB), was used. ASB represents a typical Telco application server which exercises the JVM extensively from a GC perspective, as described earlier in the paper.

Experimental Platform:

Sun Fire V480 with 4 x 900 MHz UltraSparc III processors and Solaris 8 operating system. However, the tuning techniques described here apply to other supported platforms, including Linux, Windows, and the Solaris (Intel Architecture) operating environment.

The old generation heap was sized based on the peak load for which the ASB server and the JVM were being performance tuned. The young generation heap was sized empirically by varying the young generation from a size of 12MB to 128MB. 24MB was found to be the most optimal size which gave the lowest GC sequential overhead along with acceptable GC pauses in the young generation. A young generation bigger than 24MB increased the young generation GC pauses and did not improve performance. A size smaller than 24MB significantly increased the GC sequential overhead due to an increase in GC frequency in the young generation and a premature promotion of short lived data to the old generation. Figure 4 shows the variation of GC sequential overhead, GC pauses and GC frequency in the young generation with size of the young generation heap.

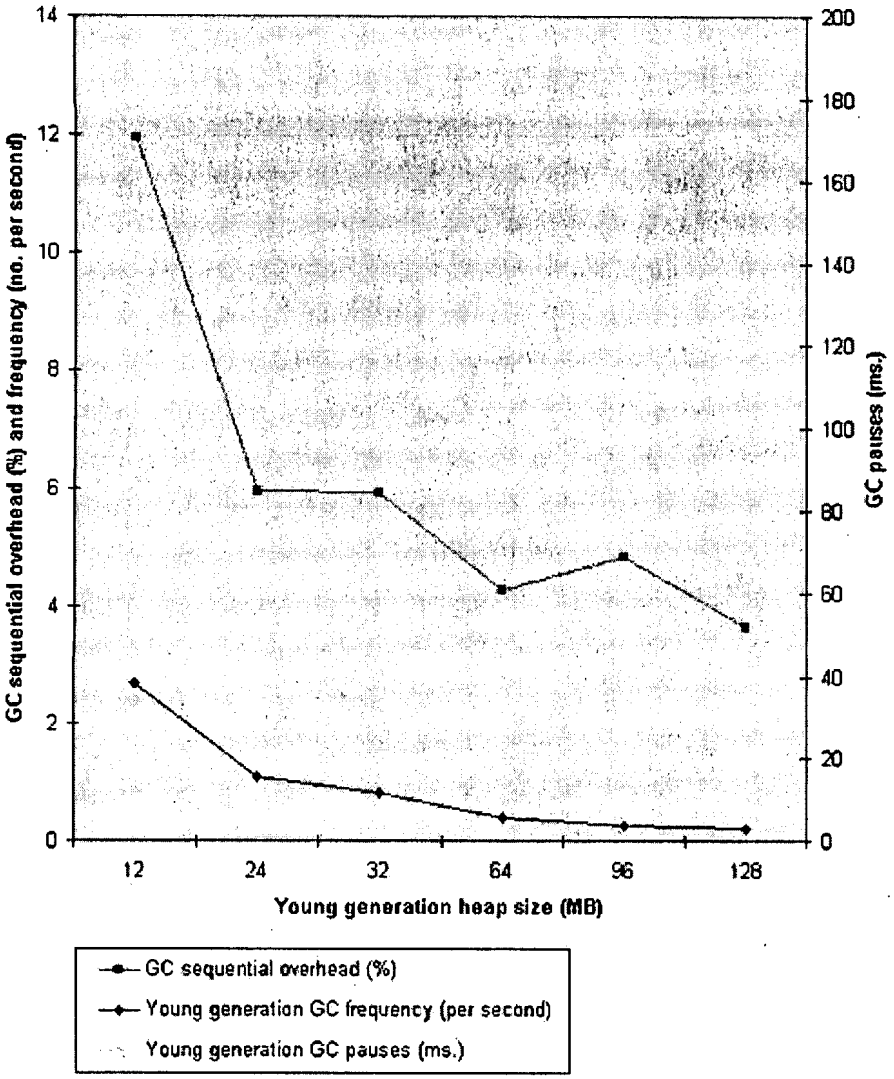

Figure 4. Variation of GC sequential overhead, GC pauses and GC frequency in the young generation with size of the young generation heap.

For an old generation of 512MB and a young generation of 24MB, results using the various garbage collection policies and other tuning techniques are shown below.

Case 1

Best results on J2SE1.4.1 using the default garbage collectors in the young and old generations for the chosen size of the JVM heap:

```
java -Xmx512m -Xms512m -XX:MaxNewSize=24m -XX:NewSize=24m -XX:SurvivorRatio=2
<application>
```

Average GC pause in old generation: 3 secs.
Average GC pause in young generation: 110 ms.
GC sequential overhead: 18.9%

Case 2

Best results on J2SE1.4.1 using the CMS collector in the old generation for the same size of the JVM heap as Case 1:

```
java -Xmx512m -Xms512m -XX:MaxNewSize=24m -XX:NewSize=24m -
XX:SurvivorRatio=128 -XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0 -
XX:CMSInitiatingOccupancyFraction=60 <application>
```

Setting `-XX:MaxTenuringThreshold=0` improved performance. Also, the value of 60 was found to be the sweet spot for `-XX:CMSInitiatingOccupancyFraction` as a value smaller than this resulted in more frequent CMS garbage collections. A value greater than this reduced the efficiency of the CMS collection.

Average GC pause in the old generation (stop-the-world `init mark and remark phase` of concurrent collection): 115 ms
Average GC pause in the young generation: 100 ms
GC sequential overhead: 8.6%

Case 3

Best results on J2SE1.4.1 using the CMS collector in the old generation and the new parallel collector in the young generation for the same size JVM heap as in Cases 1 and 2:

```
java -Xmx512m -Xms512m -XX:MaxNewSize=24m -XX:NewSize=24m -
XX:SurvivorRatio=128 -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -
XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=60 <application>
```

Average GC pause in the old generation (stop-the-world `init mark and remark phase` of concurrent collection): 142 ms
Average GC pause in the young generation: 50 ms
GC sequential overhead: 5.9%

Table 3 summarizes the three cases.

Summary Table	Average GC pause in old generation (milliseconds)	Average GC pause in young generation (milliseconds)	GC sequential overhead (%)
Default collectors	300	110	18.9 %
CMS collector	115	100	8.6%
CMS and Parallel collectors	142	50	5.9 %

Table 3 Summary of the results from the case study.

Results

The use of the CMS collector results in a nearly 2000% reduction in GC pauses in the old generation and 220% reduction in GC sequential overhead. Reduction in GC sequential overhead directly contributes to improvement in application throughput. The use of a parallel collector results in a nearly 100% reduction in GC pauses in the young generation. On a 4 CPU system, one may have expected an acceleration of 4x with the parallel collector. However, due to overhead associated with the parallel collection, the acceleration from parallel collection is not linear, and work is being done in the JVM towards refining this.

Applicable to the Enterprise

The performance tuning techniques and the new garbage collection policies described in the paper are not limited to Telco alone, and apply to Enterprise servers like Web servers, Portal servers and Application servers, as they have very similar requirements.

What Lies Ahead

Work is being done in the Sun JVMs to reduce low level command line flag tuning. Instead of choosing the sizes of the young and old generation, garbage collection policies, and so on, one may be able to specify higher level requirements in terms of maximum pauses or latency, memory footprint for the JVM, CPU utilization and so on, and the JVM would automatically set and adjust various low level parameters and policies for the application to meet the desired higher level settings.

The HotSpot JVM is being enhanced to include light weight, always-on instrumentation that provides high level performance metrics for various aspects of the operations of the JVM. It should be available in a future version of the JVM. These tools and technology are considered experimental at this time.

Java Specification Request (JSR) 174 is a specification for APIs for monitoring and management of the JVM and JSR 163 is a specification for APIs to extract profiling information from a running JVM with support for both time and memory profiling. These APIs will be designed to allow implementations which minimally perturb the profile. The APIs will allow interoperability of profiling and advanced garbage collection technologies, will allow reliable implementation on the widest range of JVMs, and will provide Java applications, system management tools and RAS-related tools with the ability to monitor the health

of the JVM as well as manage many run-time controls.

Also, work is being done on the Sun ONE Application Server 7 to meet some of the specific requirements of Telco application servers. Telco Carrier requirements are similar to the enterprise, except that, they need more 9s for service and data availability and perhaps more orchestrated service upgrades. The Sun ONE Application Server Enterprise Edition 7 contains Clustra "Always ON" technology which enables highly available state checkpointing repository as well as session and bean state. Sun ONE Application server 7 can be used to build JAIN and OSA/Parlay framework-based call processing servers. SLEE implementations can be integrated into the Sun ONE Application server to leverage the EJB container. Some down the road possibilities may also include the JSR 117 APIs for continuous availability using EJBs, and providing soft-real-time and quality-of-service capability as well as the integrated SIP stack.

Conclusions

As Java becomes increasingly popular in the Telco space, Sun JVMs are gearing to meet the challenges imposed by Telco requirements. The 64 bit JVM and larger hardware can provide solutions for configuring much larger heap sizes (not limited to 4G) to meet the need for larger servers. On a multi-CPU system, the parallel collector can reduce pauses in the young generation and concurrent collection in the old generation can hide the large GC pauses associated with the conventional old generation mark-compact collector. This in turn can bring significant reduction in GC sequential overhead, leading to much better application performance and scalabilty on multi-CPU systems.

Garbage collection in the JVM can be monitored, analyzed and tuned by mining the verbose:gc logs. The information available in the GC logs can be analyzed for sizing and configuring the JVM with switches and parameters which might be most optimal for an application's performance.

Sun ONE Application Server 7 Enterprise Edition may be suitable for implementing Telco application servers in the near future.

Acknowledgments:
The authors would like to thank the Sun JVMs garbage collection architects and experts John Coomes, Ross Knippel, Jon Masamitsu and Y.S. Ramakrishna for their help and guidance on conducting this study, providing PrintGCStats, and careful review of the paper, and their colleagues for their help.

About the Authors:
Alka Gupta is a Member of the Technical Staff at Sun Microsystems. She's responsible for working with Sun's ISVs and partners to help them adopt the emerging Sun technologies and platforms quickly and efficiently. She has been working in the area of performance tuning on Sun platforms for almost 5 years, and has been in this industry for over 8 years. Alka graduated from the Indian Institute of Technology (IIT), India.

Michael is the CTO and co-founder of Ubiquity Software Corporation. Prior to this, he worked as a consultant to a number of telecommunication companies on international OSI networks. He is an acknowledged expert in the field of network communication protocols and computer telephony integration. He has also given a series of lectures on languages and protocols to many leading UK software companies. Michael is a graduate of University College London.

References:

Application modeling and Performance tuning from Garbage Collection Perspective
The Java HotSpot Virtual Machine
Java HotSpot VM Options
Tuning Garbage Collection with the 1.3.1 Java Virtual Machine
Java and Solaris Threading
Generational Mostly-concurrent Garbage Collector
New Parallel Collector
The SIPCenter.com
Ubiquity's Application Services Broker (A Telco SIP server)
The JAIN API
Sun ONE
Sun ONE Application Server 7
JSR 174
JSR 163
Sun Docs
Clustra "Always on" technology
The Java HotSpot Virtual Machine, v1.4.1

[1] As used on this web site, the terms "Java virtual machine" or "JVM" mean a virtual machine for the Java platform.